

RabbitMQ

实战指南

朱忠华 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn

RabbitMQ

实战指南

朱忠华 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

RabbitMQ

实战指南

朱忠华 著



电子工业出版社
Publishing House of Electronics Industry
北京•BEIJING

内容简介

本书从消息中间件的概念和RabbitMQ的历史切入，主要阐述RabbitMQ的安装、使用、配置、管理、运维、原理、扩展等方面的细节。本书大致可以分为基础篇、进阶篇和高阶篇三个部分。基础篇首先介绍RabbitMQ的基本安装及使用方式，方便零基础的读者以最舒适的方式融入到RabbitMQ之中。其次介绍RabbitMQ的基本概念，包括生产者、消费者、交换器、队列、绑定等。之后通过Java语言讲述了客户端如何与RabbitMQ建立（关闭）连接、声明（删除）交换器、队列、绑定关系，以及如何发送和消费消息等。进阶篇讲述RabbitMQ的TTL、死信、延迟队列、优先级队列、RPC、消息持久化、生产端和消费端的消息确认机制等内容，以期读者能够掌握RabbitMQ的使用精髓。本书中间篇幅主要从RabbitMQ的管理、配置、运维这三个角度来为读者提供帮助文档及解决问题的思路。高阶篇主要阐述RabbitMQ的存储机制、流控及镜像队列的原理，深入地讲述RabbitMQ的一些实现细节，便于读者加深对RabbitMQ的理解。本书还涉及网络分区的概念，此内容可称为魔鬼篇，需要掌握前面的所有内容才可理解其中的门道。本书最后讲述的是RabbitMQ的一些扩展内容及附录，供读者参考之用。

本书既可供初学者学习，帮助读者了解RabbitMQ的具体细节及使用方式、原理等，也可供相关开发、测试及运维人员参考，给日常工作带来启发。

版权页

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

RabbitMQ实战指南/朱忠华著. —北京：电子工业出版社，2017.11

ISBN 978-7-121-32991-3

I. ①R... II. ①朱... III. ①JAVA语言—程序设计—指南 IV.
①TP312.8-62

中国版本图书馆CIP数据核字（2017）第264324号

责任编辑：陈晓猛

印 刷：三河市华成印务有限公司

装 订：三河市华成印务有限公司

出版发行：电子工业出版社

北京市海淀区万寿路173信箱 邮编 100036

开 本：787×9802 1/16 印张：21.75 字数：417千字

版 次：2017年11月第1版

印 次：2018年3月第2次印刷

定 价：79.00元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前言

初识RabbitMQ时，我在网上搜寻了大量的相关资料以求自己能够快速地了解它，但是这些资料零零散散而又良莠不齐。后来又寄希望于RabbitMQ的相关书籍，或许是它们都非出自国人之手，里面的陈述逻辑和案例描述都不太符合我自己的思维习惯。最后选择从头开始自研RabbitMQ，包括阅读相关源码、翻阅官网的资料以及进行大量的实验等。

平时我也有写博客的习惯，通常在工作中遇到问题时会结合所学的知识整理成文。随着一篇篇的积累，也有好几十篇的内容，渐渐地也就有了编撰成书的想法。

本书动笔之时我曾信心满满，以为能够顺其自然地完成这本书，但是写到四分之一时，发现并没有想象中的那么简单。怎样才能让理解领悟汇聚成通俗易懂的文字表达？怎样才能让书中内容前后贯通、由浅入深地阐述？有些时候可能知道怎样做、为什么这么做，而没有反思其他情形能不能做、怎样做。为了解决这些问题，我会反复对书中的内容进行迭代，对某些模糊的知识点深耕再深耕，对某些案例场景进行反复的测试，不断地完善。

在本书编写之时，我常常回想当初作为小白之时迫切地希望能够了解哪些内容，这些内容又希望以怎样的形式展现。所以本书前面几章的内容基本上是站在一个小白的视角来为读者做一个细腻的讲解，相信读者在阅读完这些内容之后能够具备合理使用RabbitMQ的能力。在后面的章节中知识点会慢慢地深入，每阅读一章的内容都会对RabbitMQ有一个更加深刻的认知。

本书中的所有内容都具备理论基础并全部实践过，书中的内容也是我在工作中的实践积累，希望本书能够让初学者对RabbitMQ有一个全面的认知，也希望有相关经验的人士可以从本书中得到一些启发，汲取一些经验。

内容大纲

本书共11章，前后章节都有相关的联系，基本上按照由浅入深、由表及里的层次逐层进行讲解。如果读者对其中的某些内容已经掌握，可以选择跳过而翻阅后面的内容，不过还是建议读者按照先后顺序进行阅读。

第1章主要针对消息中间件做一个摘要性介绍，包括什么是消息中间件、消息中间件的作用及特点等。之后引入RabbitMQ，对其历史和相关特点做一个简要概述。本章最后介绍RabbitMQ的安装及生产、消费的使用示例。

第2章主要讲述RabbitMQ的入门知识，包括生产者、消费者、队列、交换器、路由键、绑定、连接及信道等基本术语。本章还阐述了RabbitMQ与AMQP协议的对应关系。

第3章主要介绍RabbitMQ客户端开发的简单使用，按照一个生命周期对连接、创建、生产、消费及关闭等几个方面进行宏观的介绍。

第4章介绍数据可靠性的一些细节，并展示RabbitMQ的几种已具备或衍生的高级特性，包括TTL、死信队列、延迟队列、优先级队列、RPC等，这些功能在实际使用中可以让某些应用的实现变得事半功倍。

第5章主要围绕RabbitMQ管理这个主题展开，包括多租户、权限、用户、应用和集群管理、服务端状态等方面，并且从侧面讲述rabbitmqctl工具和rabbitmq_management插件的使用。

第6章主要讲述RabbitMQ的配置，以此可以通过环境变量、配置文件、运行时参数（和策略）等三种方式来定制化相应的服务。

第7章主要围绕运维层面展开论述，主要包括集群搭建、日志查看、故障恢复、集群迁移、集群监控这几个方面。

第8章主要讲述Federation和Shovel这两个插件的使用、细节及相关原理。区别于第7章中集群的部署方式，Federation和Shovel可以部署在广域网中，为RabbitMQ提供更广泛的应用空间。

第9章介绍RabbitMQ相关的一些原理，主要内容包括RabbitMQ存储机制、磁盘和内存告警、流控机制、镜像队列。了解这些实现的细节及原理十分必要，它们可以让读者在遇到问题时能够透过现象看本质。

第10章主要围绕网络分区进行展开，具体阐述网络分区的意义，如

何查看和处理网络分区，以及网络分区所带来的影响。

第11章主要探讨RabbitMQ的两个扩展内容：消息追踪及负载均衡。消息追踪可以有效地定位消息丢失的问题。负载均衡本身属于运维层面，但是负载均衡一般需要借助第三方的工具——HAProxy、LVS等实现，故本书将其视为扩展内容。

读者讨论

由于笔者水平有限，书中难免有错误之处。在本书出版后的任何时间，若你对本书有任何疑问，都可以通过zhuzhonghua.ideal@qq.com 发送邮件给笔者，也可以到笔者的个人博客<http://blog.csdn.net/u013256816>或者个人微信公众号“朱小厮的博客”中留言，向笔者阐述你的建议和想法。

致谢

首先要感谢我身处的平台，让我有机会深入地接触RabbitMQ。同时也要感谢我身边的同事，正因为有了你们的鼓励和帮助，才让我能够迅速成长，本书的问世，离不开与你们在工作中积累的点点滴滴。

感谢在我博客中提问、留言的网友，有了你们的意见和建议才能让本书更加完善。

感谢博文视点的编辑们，本书能够顺利、迅速地出版，多亏了你们的敬业精神和一丝不苟的工作态度。

感谢顾忠国、裘晟、刘松松、沈华杰、刘建刚、朱文卿、叶海民、蒋晓峰等朋友的第一波支持。

感谢Happy Sunshine Boy、苏逊、文斌、方志斌、codeOfQuite、翁庭贵等朋友对本书的勘误提出的宝贵意见。

最后还要感谢我的家人，在我占用绝大部分的业余时间进行写作的时候，能够给予我极大的宽容、理解和支持，让我能够全身心地投入到写作之中。

朱忠华

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- 提交勘误：您对书中内容的修改意见可在提交勘误处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。

- 交流互动：在页面下方 读者评论 处留下您的疑问或观点，与其他读者一同学习交流。

页面入口：<http://www.broadview.com.cn/32991>



目 录

内容简介

版权页

前言

第1章 RabbitMQ简介

1.1 什么是消息中间件

1.2 消息中间件的作用

1.3 RabbitMQ的起源

1.4 RabbitMQ的安装及简单使用

1.4.1 安装Erlang

1.4.2 RabbitMQ的安装

1.4.3 RabbitMQ的运行

1.4.4 生产和消费消息

1.5 小结

第2章 RabbitMQ入门

2.1 相关概念介绍

2.1.1 生产者和消费者

2.1.2 队列

2.1.3 交换器、路由键、绑定

2.1.4 交换器类型

2.1.5 RabbitMQ运转流程

2.2 AMQP协议介绍

2.2.1 AMQP生产者流转过程

2.2.2 AMQP 消费者流转过程

2.2.3 AMQP命令概览

2.3 小结

第3章 客户端开发向导

3.1 连接RabbitMQ

3.2 使用交换器和队列

3.2.1 exchangeDeclare方法详解

3.2.2 queueDeclare方法详解

3.2.3 queueBind方法详解

3.2.4 exchangeBind方法详解

3.2.5 何时创建

3.3 发送消息

3.4 消费消息

3.4.1 推模式

3.4.2 拉模式

3.5 消费端的确认与拒绝

3.6 关闭连接

3.7 小结

第4章 RabbitMQ进阶

4.1 消息何去何从

4.1.1 mandatory参数

4.1.2 immediate参数

4.1.3 备份交换器

4.2 过期时间（TTL）

4.2.1 设置消息的TTL

4.2.2 设置队列的TTL

4.3 死信队列

4.4 延迟队列

4.5 优先级队列

4.6 RPC实现

4.7 持久化

4.8 生产者确认

4.8.1 事务机制

4.8.2 发送方确认机制

4.9 消费端要点介绍

- 4.9.1 消息分发
- 4.9.2 消息顺序性
- 4.9.3 弃用QueueingConsumer

4.10 消息传输保障

4.11 小结

第5章 RabbitMQ管理

5.1 多租户与权限

5.2 用户管理

5.3 Web端管理

5.4 应用与集群管理

5.4.1 应用管理

5.4.2 集群管理

5.5 服务端状态

5.6 HTTP API接口管理

5.7 小结

第6章 RabbitMQ配置

6.1 环境变量

6.2 配置文件

6.2.1 配置项

6.2.2 配置加密

6.2.3 优化网络配置

6.3 参数及策略

6.4 小结

第7章 RabbitMQ运维

7.1 集群搭建

7.1.1 多机多节点配置

7.1.2 集群节点类型

7.1.3 剔除单个节点

7.1.4 集群节点的升级

7.1.5 单机多节点配置

7.2 查看服务日志

7.3 单节点故障恢复

7.4 集群迁移

7.4.1 元数据重建

7.4.2 数据迁移和客户端连接的切换

7.4.3 自动化迁移

7.5 集群监控

7.5.1 通过HTTP API接口提供监控数据

7.5.2 通过客户端提供监控数据

7.5.3 检测RabbitMQ服务是否健康

7.5.4 元数据管理与监控

7.6 小结

第8章 跨越集群的界限

8.1 Federation

8.1.1 联邦交换器

8.1.2 联邦队列

8.1.3 Federation的使用

8.2 Shovel

8.2.1 Shovel的原理

8.2.2 Shovel的使用

8.2.3 案例：消息堆积的治理

8.3 小结

第9章 RabbitMQ高阶

9.1 存储机制

9.1.1 队列的结构

9.1.2 惰性队列

9.2 内存及磁盘告警

9.2.1 内存告警

9.2.2 磁盘告警

9.3 流控

9.3.1 流控的原理

9.3.2 案例：打破队列的瓶颈

9.4 镜像队列

9.5 小结

第10章 网络分区

10.1 网络分区的意义

10.2 网络分区的判定

10.3 网络分区的模拟

10.4 网络分区的影响

10.4.1 未配置镜像

10.4.2 已配置镜像

10.5 手动处理网络分区

10.6 自动处理网络分区

10.6.1 pause-minority模式

10.6.2 pause-if-all-down模式

10.6.3 autoheal模式

10.6.4 挑选哪种模式

10.7 案例：多分区情形

10.8 小结

第11章 RabbitMQ扩展

11.1 消息追踪

11.1.1 Firehose

11.1.2 rabbitmq_tracing插件

11.1.3 案例：可靠性检测

11.2 负载均衡

11.2.1 客户端内部实现负载均衡

11.2.2 使用HAProxy实现负载均衡

11.2.3 使用Keepalived实现高可靠负载均衡

11.2.4 使用Keepalived+LVS实现负载均衡

11.3 小结

附录A 集群元数据信息示例

附录B /api/nodes接口详细内容

附录C 网络分区图谱

第1章 RabbitMQ简介

RabbitMQ是目前非常热门的一款消息中间件，不管是互联网行业还是传统行业都在大量地使用。RabbitMQ凭借其高可靠、易扩展、高可用及丰富的功能特性受到越来越多企业的青睐。作为一个合格的开发者，有必要深入地了解RabbitMQ的相关知识，为自己的职业生涯添砖加瓦。

1.1 什么是消息中间件

消息（Message）是指在应用间传送的数据。消息可以非常简单，比如只包含文本字符串、JSON等，也可以很复杂，比如内嵌对象。

消息队列中间件（Message Queue Middleware，简称为MQ）是指利用高效可靠的消息传递机制进行与平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信。

消息队列中间件，也可以称为消息队列或者消息中间件。它一般有两种传递模式：点对点（P2P，Point-to-Point）模式和发布/订阅

（Pub/Sub）模式。点对点模式是基于队列的，消息生产者发送消息到队列，消息消费者从队列中接收消息，队列的存在使得消息的异步传输成为可能。发布订阅模式定义了如何向一个内容节点发布和订阅消息，这个内容节点称为主题（topic），主题可以认为是消息传递的中介，消息发布者将消息发布到某个主题，而消息订阅者则从主题中订阅消息。主题使得消息的订阅者与消息的发布者互相保持独立，不需要进行接触即可保证消息的传递，发布/订阅模式在消息的一对多广播时采用。

目前开源的消息中间件有很多，比较主流的有RabbitMQ、Kafka、ActiveMQ、RocketMQ等。面向消息的中间件（简称为MOM，Message Oriented Middleware）提供了以松散耦合的灵活方式集成应用程序的一种机制。它们提供了基于存储和转发的应用程序之间的异步数据发送，即应用程序彼此不直接通信，而是与作为中介的消息中间件通信。消息中间件提供了有保证的消息发送，应用程序开发人员无须了解远程过程调用（RPC）和网络通信协议的细节。

消息中间件适用于需要可靠的数据传送的分布式环境。采用消息中间件的系统中，不同的对象之间通过传递消息来激活对方的事件，以完成相应的操作。发送者将消息发送给消息服务器，消息服务器将消息存放在若干队列中，在合适的时候再将消息转发给接收者。消息中间件能在不同平台之间通信，它常被用来屏蔽各种平台及协议之间的特性，实现应用程序之间的协同，其优点在于能够在客户和服务器之间提供同步

和异步的连接，并且在任何时刻都可以将消息进行传送或者存储转发，这也是它比远程过程调用更进步的原因。

举例说明，如图1-1所示，应用程序A与应用程序B通过使用消息中间件的应用程序编程接口（API，Application Program Interface）发送消息来进行通信。

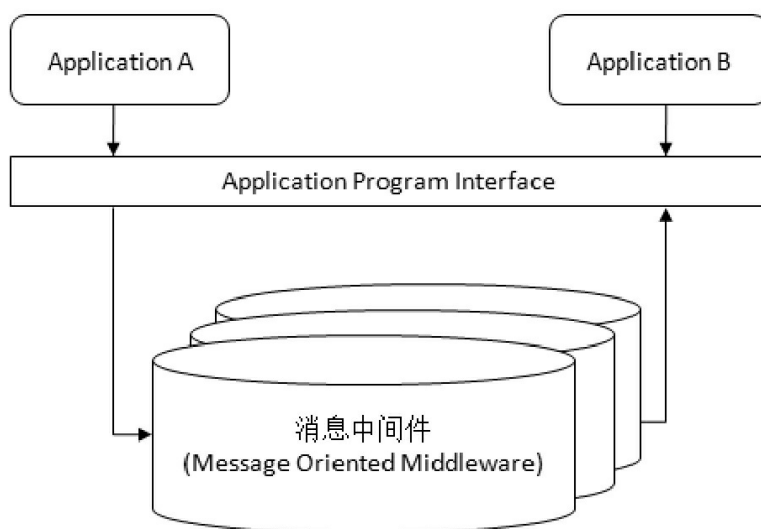


图1-1 应用通过消息中间件进行通信

消息中间件将消息路由给应用程序B，这样消息就可存在于完全不同的计算机上。消息中间件负责处理网络通信，如果网络连接不可用，消息中间件会存储消息，直到连接变得可用，再将消息转发给应用程序B。灵活性的另一方面体现在，当应用程序A发送其消息时，应用程序B甚至可以处于不运行状态，消息中间件将保留这份消息，直到应用程序B开始执行并消费消息，这样还防止了应用程序A因为等待应用程序B消费消息而出现阻塞。这种异步通信方式要求应用程序的设计与现在大多数应用不同。不过对于时间无关或并行处理的场景，它可能是一个极其有用的方法。

1.2 消息中间件的作用

消息中间件凭借其独到的特性，在不同的应用场景下可以展现不同的作用。总的来说，消息中间件的作用可以概括如下。

解耦： 在项目启动之初来预测将来会碰到什么需求是极其困难的。消息中间件在处理过程中插入了一个隐含的、基于数据的接口层，两边的处理过程都要实现这一接口，这允许你独立地扩展或修改两边的处理过程，只要确保它们遵守同样的接口约束即可。

冗余（存储）： 有些情况下，处理数据的过程会失败。消息中间件可以把数据进行持久化直到它们已经被完全处理，通过这一方式规避了数据丢失风险。在把一个消息从消息中间件中删除之前，需要你的处理系统明确地指出该消息已经被处理完成，从而确保你的数据被安全地保存直到你使用完毕。

扩展性： 因为消息中间件解耦了应用的处理过程，所以提高消息入队和处理的效率是很容易的，只要另外增加处理过程即可，不需要改变代码，也不需要调节参数。

削峰： 在访问量剧增的情况下，应用仍然需要继续发挥作用，但是这样的突发流量并不常见。如果以能处理这类峰值为标准而投入资源，无疑是巨大的浪费。使用消息中间件能够使关键组件支撑突发访问压力，不会因为突发的超负荷请求而完全崩溃。

可恢复性： 当系统一部分组件失效时，不会影响到整个系统。消息中间件降低了进程间的耦合度，所以即使一个处理消息的进程挂掉，加入消息中间件中的消息仍然可以在系统恢复后进行处理。

顺序保证： 在大多数使用场景下，数据处理的顺序很重要，大部分消息中间件支持一定程度上的顺序性。

缓冲： 在任何重要的系统中，都会存在需要不同处理时间的元素。消息中间件通过一个缓冲层来帮助任务最高效率地执行，写入消息中间件的处理会尽可能快速。该缓冲层有助于控制和优化数据流经过系统的速度。

异步通信： 在很多时候应用不想也不需要立即处理消息。消息中间件提供了异步处理机制，允许应用把一些消息放入消息中间件中，但并不立即处理它，在之后需要的时候再慢慢处理。

1.3 RabbitMQ的起源

RabbitMQ是采用Erlang语言实现AMQP（Advanced Message Queuing Protocol，高级消息队列协议）的消息中间件，它最初起源于金融系统，用于在分布式系统中存储转发消息。

在此之前，有一些消息中间件的商业实现，比如微软的MSMQ（MicroSoft Message Queue）、IBM的WebSphere等。由于高昂的价格，一般只应用于大型组织机构，它们需要可靠性、解耦及实时消息通信的功能。由于商业壁垒，商业MQ供应商想要解决应用互通的问题，而不是去创建标准来实现不同的MQ产品间的互通，或者允许应用程序更改MQ平台。

为了打破这个壁垒，同时为了能够让消息在各个消息队列平台间互通，JMS（Java Message Service）应运而生。JMS试图通过提供公共Java API的方式，隐藏单独MQ产品供应商提供的实际接口，从而跨越了壁垒，以及解决了互通问题。从技术上讲，Java应用程序只需针对JMS API编程，选择合适的MQ驱动即可，JMS会打理好其他部分。ActiveMQ就是JMS的一种实现。不过尝试使用单独标准化接口来胶合众多不同的接口，最终会暴露出问题，使得应用程序变得更加脆弱。所以急需一种新的消息通信标准化方案。

在2006年6月，由Cisco、Redhat、iMatix等联合制定了AMQP的公开标准，由此AMQP登上了历史的舞台。它是应用层协议的一个开放标准，以解决众多消息中间件的需求和拓扑结构问题。它为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言等条件的限制。

RabbitMQ最初版本实现了AMQP的一个关键特性：使用协议本身就可以对队列和交换器（Exchange）这样的资源进行配置。对于商业MQ供应商来说，资源配置需要通过管理终端的特定工具才能完成。RabbitMQ的资源配置能力使其成为构建分布式应用的最完美的通信总线，特别有助于充分利用基于云的资源 and 进行快速开发。

RabbitMQ是由RabbitMQ Technologies Ltd开发并且提供商业支持

的。取Rabbit这样一个名字，是因为兔子行动非常迅速且繁殖起来非常疯狂，RabbitMQ的开创者认为以此命名这个分布式软件再合适不过了。RabbitMQ Technologies Ltd在2010年4月被SpringSource（VMWare的一个部门）收购，在2013年5月并入Pivotal，其实VMWare、Pivotal和EMC本质上是一家。不同的是VMWare是独立上市子公司，而Pivotal是整合了EMC的某些资源，现在并没有上市。至今你也可以在RabbitMQ的官网^[1]上的Logo旁看到“by Pivotal”的字样，如图1-2所示。



图1-2 官网Logo

RabbitMQ发展到今天，被越来越多的人认可，这和它在易用性、扩展性、可靠性和高可用性等方面的卓著表现是分不开的。RabbitMQ的具体特点可以概括为以下几点。

- ✧ 可靠性：RabbitMQ使用一些机制来保证可靠性，如持久化、传输确认及发布确认等。

- ✧ 灵活的路由：在消息进入队列之前，通过交换器来路由消息。对于典型的路由功能，RabbitMQ已经提供了一些内置的交换器来实现。针对更复杂的路由功能，可以将多个交换器绑定在一起，也可以通过插件机制来实现自己的交换器。

- ✧ 扩展性：多个RabbitMQ节点可以组成一个集群，也可以根据实际业务情况动态地扩展集群中节点。

- ✧ 高可用性：队列可以在集群中的机器上设置镜像，使得在部分节点出现问题的情况下队列仍然可用。

- ✧ 多种协议：RabbitMQ除了原生支持AMQP协议，还支持STOMP、MQTT等多种消息中间件协议。

- ✧ 多语言客户端：RabbitMQ几乎支持所有常用语言，比如Java、Python、Ruby、PHP、C#、JavaScript等。

- ✧ 管理界面：RabbitMQ提供了一个易用的用户界面，使得用户可以监控和管理消息、集群中的节点等。

✧ 插件机制：RabbitMQ提供了许多插件，以实现从多方面进行扩展，当然也可以编写自己的插件。

1.4 RabbitMQ的安装及简单使用

这里首先介绍RabbitMQ的安装过程，然后演示发送和消费消息的具体实现，以期让读者对RabbitMQ有比较直观的感受。

前面提到了RabbitMQ是由Erlang语言编写的，也正因如此，在安装RabbitMQ之前需要安装Erlang。建议采用较新版的Erlang，这样可以获得较多更新和改进，可以到官网（<http://www.erlang.org/downloads>）下载。截止本书撰稿，最新版本为20.0，本书示例大多采用19.x的版本。

本书如无特指，所有程序都是在Linux下运行的，毕竟RabbitMQ大多部署在Linux操作系统之中。

1.4.1 安装Erlang

下面首先演示Erlang的安装。第一步，解压安装包，并配置安装目录，这里我们预备安装到/opt/erlang目录下：

```
[root@hidden ~]# tar zxvf otp_src_19.3.tar.gz
```

```
[root@hidden ~]# cd otp_src_19.3
```

```
[root@hidden otp_src_19.3]# ./configure --prefix=/opt/erlang
```

第二步，如果出现类似关键报错信息：No curses library functions found。那么此时需要安装ncurses，安装步骤（遇到提示输入y后直接回车即可）如下：

```
[root@hidden otp_src_19.3]# yum install ncurses-devel
```

第三步，安装Erlang：

```
[root@hidden otp_src_19.3]# make
```

```
[root@hidden otp_src_19.3]# make install
```

如果在安装的过程中出现类似“No ***** found”的提示，可根据提示信息安装相应的包，之后再执行第二或者第三步，直到提示安装完毕

为止。

第四步，修改/etc/profile配置文件，添加下面的环境变量：

```
ERLANG_HOME=/opt/erlang
```

```
export PATH=$PATH:$ERLANG_HOME/bin
```

```
export ERLANG_HOME
```

最后执行如下命令让配置文件生效：

```
[root@hidden otp_src_19.3]# source /etc/profile
```

可以输入erl命令来验证Erlang是否安装成功，如果出现类似以下的提示即表示安装成功：

```
[root@hidden ~]# erl
```

```
Erlang/OTP 19 [erts-8.1] [source] [64-bit] [smp:4:4] [async-threads:10]  
[hipe] [kernel-poll:false]
```

```
Eshell V8.1 (abort with ^G)
```

```
1>
```

1.4.2 RabbitMQ的安装

RabbitMQ的安装比Erlang的安装要简单，直接将下载的安装包解压到相应的目录下即可，官网下载地址：

<http://www.rabbitmq.com/releases/rabbitmq-server/>。本书撰稿时的最新版本为3.6.12，本书示例大多采用同一系列的3.6.x版本。

这里选择将RabbitMQ安装到与Erlang同一个目录（/opt）下面：

```
[root@hidden ~]# tar zxvf rabbitmq-server-generic-unix-3.6.10.tar.gz -  
C /opt
```

```
[root@hidden ~]# cd /opt
```

```
[root@hidden ~]# mv rabbitmq_server-3.6.10 rabbitmq
```

同样修改/etc/profile文件，添加下面的环境变量：

```
export PATH=$PATH:/opt/rabbitmq/sbin
```

```
export RABBITMQ_HOME=/opt/rabbitmq
```

之后执行source/etc/profile命令让配置文件生效。

1.4.3 RabbitMQ的运行

在修改了/etc/profile配置文件之后，可以任意打开一个Shell窗口，输入如下命令以运行RabbitMQ服务：

```
rabbitmq-server -detached
```

在rabbitmq-server命令后面添加一个“-detached”参数是为了能够让RabbitMQ服务以守护进程的方式在后台运行，这样就不会因为当前Shell窗口的关闭而影响服务。

运行rabbitmqctl status命令查看RabbitMQ是否正常启动，示例如下：

```
[root@hidden ~]# rabbitmqctl status
Status of node rabbit@hidden
[{pid,6458},
 {running_applications,
  [{rabbitmq_management,"RabbitMQ Management Console","3.6.10"}],
```

```

{rabbitmq_management_agent,"RabbitMQ Management Agent","3.6.10"},
{rabbitmq_web_dispatch,"RabbitMQ Web Dispatcher","3.6.10"},
{rabbit,"RabbitMQ","3.6.10"},
{mnesia,"MNESIA CXC 138 12","4.14.1"},
{amqp_client,"RabbitMQ AMQP Client","3.6.10"},
{os_mon,"CPO CXC 138 46","2.4.1"},
{rabbit_common,
  "Modules shared by rabbitmq-server and rabbitmq-erlang-client",
  "3.6.10"},
{compiler,"ERTS CXC 138 10","7.0.2"},
{inets,"INETS CXC 138 49","6.3.3"},
{cowboy,"Small, fast, modular HTTP server.","1.0.4"},
{ranch,"Socket acceptor pool for TCP protocols.","1.3.0"},
{ssl,"Erlang/OTP SSL application","8.0.2"},
{public_key,"Public key infrastructure","1.2"},
{cowlib,"Support library for manipulating Web protocols.","1.0.2"},
{crypto,"CRYPTO","3.7.1"},
{syntax_tools,"Syntax tools","2.1"},
{asn1,"The Erlang ASN1 compiler version 4.0.4","4.0.4"},
{xmerl,"XML parser","1.3.12"},
{sasl,"SASL CXC 138 11","3.0.1"},
{stdlib,"ERTS CXC 138 10","3.1"},
{kernel,"ERTS CXC 138 10","5.1"]}],
{os,{unix,linux}},
{erlang_version,
  "Erlang/OTP 19 [erts-8.1] [source] [64-bit] [smp:4:4]
[async-threads:64] [hipe] [kernel-poll:true]\n"},
{memory,
  [{total,61061688},
   {connection_readers,0},
   {connection_writers,0},
   {connection_channels,0},
   {connection_other,2832},
   {queue_procs,2832},
   {queue_slave_procs,0},
   {plugins,487104},
   {other_proc,21896528},
   {mnesia,60800},
   {metrics,193616},
   {mgmt_db,137720},
   {msg_index,43392},
   {other_ets,2485240},
   {binary,132984},
   {code,24661210},
   {atom,1033401},
   {other_system,10114813}]},
{alarms,[]},
{listeners,[{clustering,25672,"::"},{amqp,5672,"::"},{http,15672,"::"}]},
{vm_memory_high_watermark,0.4},

```

```
{vm_memory_limit,3301929779},
{disk_free_limit,500000000},
{disk_free,30244855808},
{file_descriptors,
  [{total_limit,924},{total_used,2},{sockets_limit,829},{sockets_used,0}]},
{processes,[{limit,1048576},{used,323}]},
{run_queue,0},
{uptime,11},
{kernel,{net_ticktime,60}}]
```

如果RabbitMQ正常启动，会输出如上所示的信息。当然也可以通过rabbitmqctl cluster_status命令来查看集群信息，目前只有一个RabbitMQ服务节点，可以看作单节点的集群：

```
[root@hidden ~]# rabbitmqctl cluster_status
```

```
Cluster status of node rabbit@hidden
```

```
[{nodes,[{disc,[rabbit@hidden]}]},
 {running_nodes,[rabbit@hidden]},
 {cluster_name,<<"rabbit@hidden">>},
 {partitions,[],},
 {alarms,[{rabbit@hidden,[],}]}]
```

在后面的7.1节中会对多节点的集群配置进行介绍。

1.4.4 生产和消费消息

本节将演示如何使用RabbitMQ Java客户端生产和消费消息。本书中如无特殊说明，示例都采用Java语言来演示，包括RabbitMQ官方文档基本上也是采用Java语言来进行演示的。当然如前面所提及的，RabbitMQ客户端可以支持很多种语言。

目前最新的RabbitMQ Java客户端版本为4.2.1，相应的maven构建文件如下：

```
<!-- https://mvnrepository.com/artifact/com.rabbitmq/amqp-client -->
<dependency>
  <groupId>com.rabbitmq</groupId>
  <artifactId>amqp-client</artifactId>
  <version>4.2.1</version>
</dependency>
```

读者可以根据项目的实际情况进行调节。

默认情况下，访问RabbitMQ服务的用户名和密码都是“guest”，这个账户有限制，默认只能通过本地网络（如localhost）访问，远程网络访问受限，所以在实现生产和消费消息之前，需要另外添加一个用户，并设置相应的访问权限。

添加新用户，用户名为“root”，密码为“root123”：

```
[root@hidden ~]# rabbitmqctl add_user root root123
```

Creating user "root" ...

为root用户设置所有权限：

```
[root@hidden ~]# rabbitmqctl set_permissions -p / root ".*" ".*" ".*"
```

Setting permissions for user "root" in vhost "/" ...

设置root用户为管理员角色：

```
[root@hidden ~]# rabbitmqctl set_user_tags root administrator
```

Setting tags for user "root" to [administrator] ...

如果读者在使用RabbitMQ的过程中遇到类似如下的报错，那么很可能就是账户管理的问题，需要根据上面的步骤进行设置，之后再运行程序。

```
Exception in thread "main"
com.rabbitmq.client.AuthenticationFailureException: ACCESS_REFUSED -
Login was refused using authentication mechanism PLAIN. For details see
the broker logfile.
```

计算机的世界是从“Hello World！”开始的，这里我们也沿用惯例，首先生产者发送一条消息“Hello World！”至RabbitMQ中，之后由消费者

消费。下面先演示生产者客户端的代码（代码清单1-1），接着再演示消费者客户端的代码（代码清单1-2）。

代码清单1-1 生产者客户端代码

```
package com.zzh.rabbitmq.demo;

import com.rabbitmq.client.Channel;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import com.rabbitmq.client.MessageProperties;
import java.io.IOException;
import java.util.concurrent.TimeoutException;

public class RabbitProducer {
    private static final String EXCHANGE_NAME = "exchange_demo";
    private static final String ROUTING_KEY = "routingkey_demo";
    private static final String QUEUE_NAME = "queue_demo";
    private static final String IP_ADDRESS = "192.168.0.2";
    private static final int PORT = 5672; // RabbitMQ 服务端默认端口号为 5672
```

```

public static void main(String[] args) throws IOException,
    TimeoutException, InterruptedException {
    ConnectionFactory factory = new ConnectionFactory();
    factory.setHost(IP_ADDRESS);
    factory.setPort(PORT);
    factory.setUsername("root");
    factory.setPassword("root123");
    Connection connection = factory.newConnection();//创建连接
    Channel channel = connection.createChannel();//创建信道
    //创建一个type="direct"、持久化的、非自动删除的交换器
    channel.exchangeDeclare(EXCHANGE_NAME, "direct", true, false, null);
    //创建一个持久化、非排他的、非自动删除的队列
    channel.queueDeclare(Queue_NAME, true, false, false, null);
    //将交换器与队列通过路由键绑定
    channel.queueBind(Queue_NAME, EXCHANGE_NAME, ROUTING_KEY);
    //发送一条持久化的消息: hello world!
    String message = "Hello World!";
    channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
        MessageProperties.PERSISTENT_TEXT_PLAIN,
        message.getBytes());
    //关闭资源
    channel.close();
    connection.close();
}
}

```

为了方便初学者能够正确地运行本段代码，完成“新手上路”的任务，这里将一个完整的程序展示出来。在后面的章节中，如无特别需要，都只会展示出部分关键代码。

上面的生产者客户端的代码首先和RabbitMQ服务器建立一个连接（**Connection**），然后在这个连接之上创建一个信道（**Channel**）。之后创建一个交换器（**Exchange**）和一个队列（**Queue**），并通过路由键进行绑定（在2.1节中会有关于交换器、队列及路由键的详细解释）。然后发送一条消息，最后关闭资源。

代码清单1-2 消费者客户端代码

```
package com.zzh.rabbitmq.demo;

import com.rabbitmq.client.*;
import java.io.IOException;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;

public class RabbitConsumer {
    private static final String QUEUE_NAME = "queue_demo";
```

```

private static final String IP_ADDRESS = "192.168.0.2";
private static final int PORT = 5672;

public static void main(String[] args) throws IOException,
    TimeoutException, InterruptedException {
    Address[] addresses = new Address[]{
        new Address(IP_ADDRESS, PORT)
    };
    ConnectionFactory factory = new ConnectionFactory();
    factory.setUsername("root");
    factory.setPassword("root123");
    //这里的连接方式与生产者的 demo 略有不同, 注意辨别区别
    Connection connection = factory.newConnection(addresses); //创建连接
    final Channel channel = connection.createChannel(); //创建信道
    channel.basicQos(64); //设置客户端最多接收未被 ack 的消息的个数
    Consumer consumer = new DefaultConsumer(channel) {
        @Override
        public void handleDelivery(String consumerTag,
            Envelope envelope,
            AMQP.BasicProperties properties,
            byte[] body)
            throws IOException {
            System.out.println("recv message: " + new String(body));
            try {
                TimeUnit.SECONDS.sleep(1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            channel.basicAck(envelope.getDeliveryTag(), false);
        }
    };
    channel.basicConsume(QUEUE_NAME, consumer);
    //等待回调函数执行完毕之后, 关闭资源
    TimeUnit.SECONDS.sleep(5);
    channel.close();
    connection.close();
}
}

```

注意这里采用的是继承DefaultConsumer的方式来实现消费, 有过RabbitMQ使用经验的读者也许会喜欢采用QueueingConsumer的方式来实现消费, 但是我们并不推荐, 使用QueueingConsumer会有一些隐患。同时, 在RabbitMQ Java客户端4.0.0版本开始将QueueingConsumer标记

为@Deprecated，在后面的大版本中会删除这个类，更多详细内容可以参考4.9.3节。

通过上面的演示，相信各位读者对RabbitMQ有了一个初步的认识。但是这也仅仅是个开始，路漫漫其修远兮，愿君能上下而求索。

1.5 小结

本章首先针对消息中间件做了一个摘要性的介绍，包括什么是消息中间件、消息中间件的作用及消息中间件的特点等。之后引入RabbitMQ，对其历史做一个简单的阐述，比如RabbitMQ具备哪些特点。本章后面的篇幅介绍了RabbitMQ的安装及简单使用，通过演示生产者生产消息，以及消费者消费消息来给读者一个对于RabbitMQ的最初的印象，为后面的探索过程打下基础。

第2章 RabbitMQ入门

第1章的内容让我们对消息中间件和RabbitMQ本身有了大致的印象，但这是最浅显的。为了能够撬开RabbitMQ的大门，还需要针对RabbitMQ本身及其所遵循的AMQP协议中的一些细节做进一步的探究。在阅读本章内容的时候可以带着这样的一些疑问：RabbitMQ的模型架构是什么？AMQP协议又是什么？这两者之间又有何种紧密的关系？消息从生产者发出到消费者消费这一过程中要经历一些什么？

2.1 相关概念介绍

RabbitMQ整体上是一个生产者与消费者模型，主要负责接收、存储和转发消息。可以把消息传递的过程想象成：当你将一个包裹送到邮局，邮局会暂存并最终将邮件通过邮递员送到收件人的手上，RabbitMQ就好比由邮局、邮箱和邮递员组成的一个系统。从计算机术语层面来说，RabbitMQ模型更像是一种交换机模型。

RabbitMQ的整体模型架构如图2-1所示。

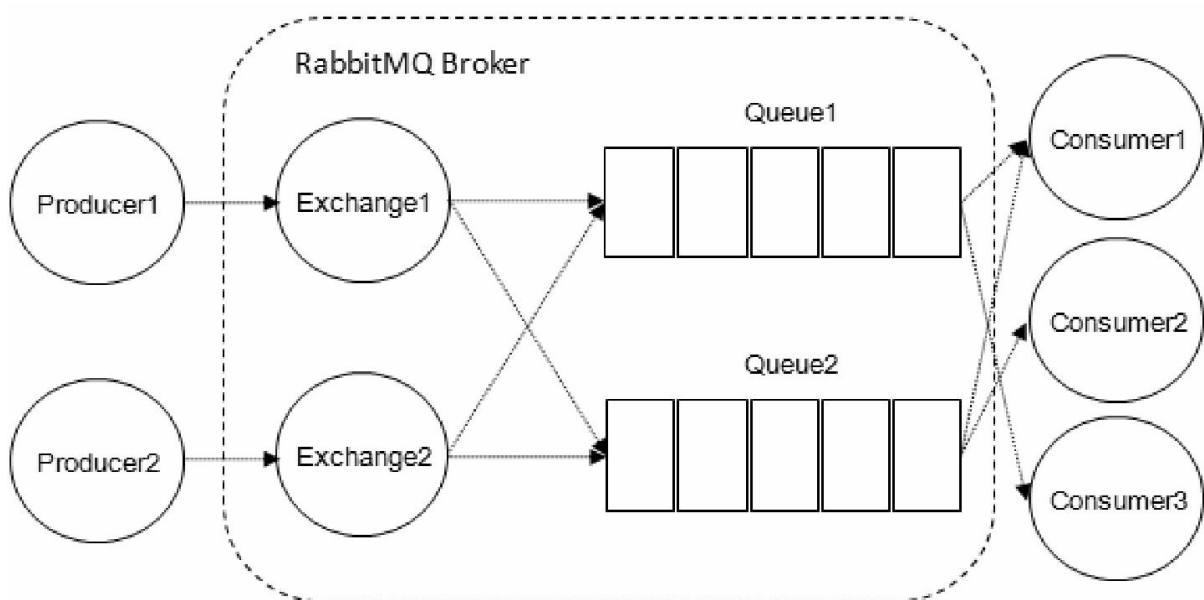


图2-1 RabbitMQ的模型架构

2.1.1 生产者和消费者

Producer: 生产者，就是投递消息的一方。

生产者创建消息，然后发布到RabbitMQ中。消息一般可以包含2个部分：消息体和标签（Label）。消息体也可以称之为payload，在实际应用中，消息体一般是一个带有业务逻辑结构的数据，比如一个JSON字符串。当然可以进一步对这个消息体进行序列化操作。消息的标签用来表述这条消息，比如一个交换器的名称和一个路由键。生产者把消息

交由RabbitMQ，RabbitMQ之后会根据标签把消息发送给感兴趣的消费者（Consumer）。

Consumer：消费者，就是接收消息的一方。

消费者连接到RabbitMQ服务器，并订阅到队列上。当消费者消费一条消息时，只是消费消息的消息体（payload）。在消息路由的过程中，消息的标签会丢弃，存入到队列中的消息只有消息体，消费者也只会消费到消息体，也就不知道消息的生产者是谁，当然消费者也不需要知道。

Broker：消息中间件的服务节点。

对于RabbitMQ来说，一个RabbitMQ Broker可以简单地看作一个RabbitMQ服务节点，或者RabbitMQ服务实例。大多数情况下也可以将一个RabbitMQ Broker看作一台RabbitMQ服务器。

图2-2展示了生产者将消息存入RabbitMQ Broker，以及消费者从Broker中消费数据的整个流程。

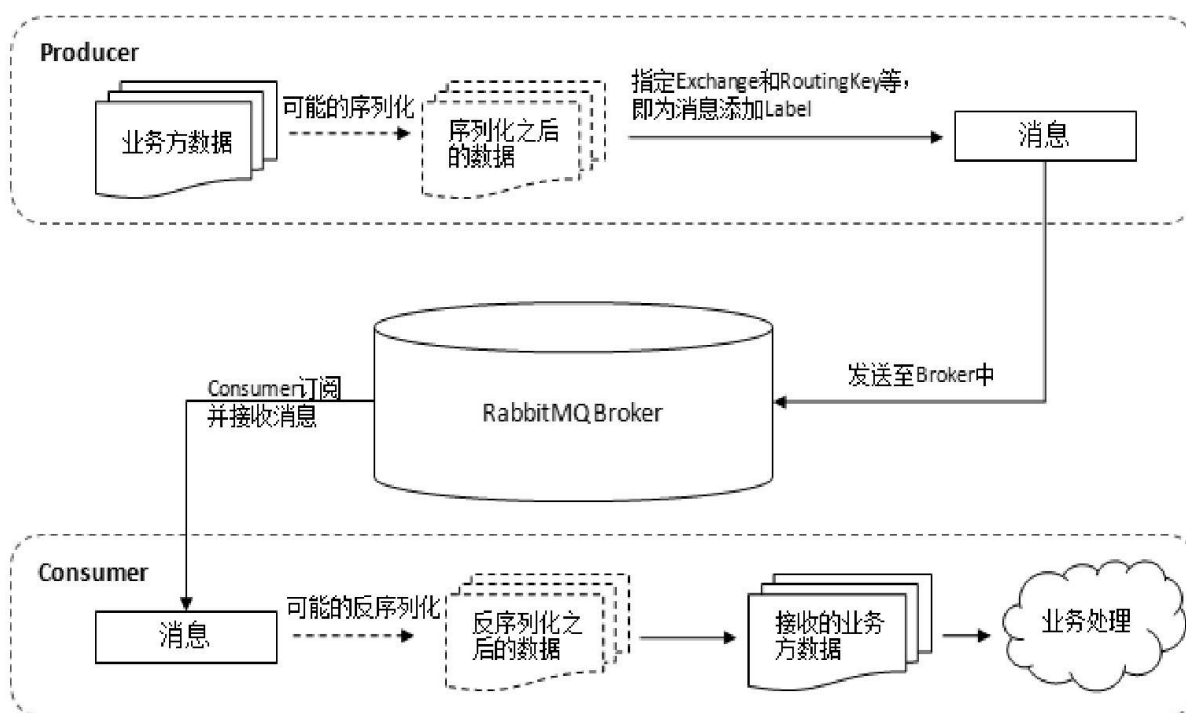


图2-2 消息队列的运转过程

首先生产者将业务方数据进行可能的包装，之后封装成消息，发送

（AMQP协议里这个动作对应的命令为Basic.Publish）到Broker中。消费者订阅并接收消息（AMQP协议里这个动作对应的命令为Basic.Consume或者Basic.Get），经过可能的解包处理得到原始的数据，之后再进行处理逻辑。这个业务处理逻辑并不一定需要和接收消息的逻辑使用同一个线程。消费者进程可以使用一个线程去接收消息，存入到内存中，比如使用Java中的BlockingQueue。业务处理逻辑使用另一个线程从内存中读取数据，这样可以将应用进一步解耦，提高整个应用的处理效率。

2.1.2 队列

Queue: 队列，是RabbitMQ的内部对象，用于存储消息。参考图2-1，队列可以用图2-3表示。

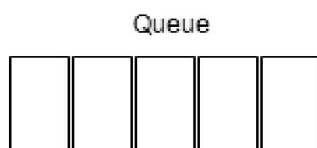


图2-3 队列

RabbitMQ中消息都只能存储在队列中，这一点和Kafka这种消息中间件相反。Kafka将消息存储在topic（主题）这个逻辑层面，而相对应的队列逻辑只是topic实际存储文件中的位移标识。RabbitMQ的生产者生产消息并最终投递到队列中，消费者可以从队列中获取消息并消费。

多个消费者可以订阅同一个队列，这时队列中的消息会被平均分摊（Round-Robin，即轮询）给多个消费者进行处理，而不是每个消费者都收到所有的消息并处理，如图2-4所示。

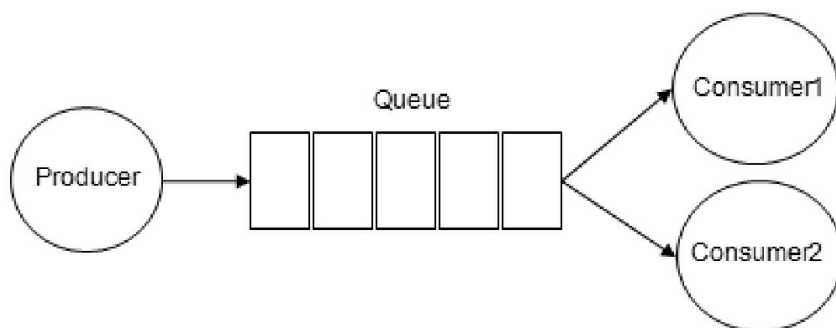


图2-4 多个消费者

RabbitMQ不支持队列层面的广播消费，如果需要广播消费，需要在其上进行二次开发，处理逻辑会变得异常复杂，同时也不建议这么做。

2.1.3 交换器、路由键、绑定

Exchange: 交换器。在图2-4中我们暂时可以理解成生产者将消息投递到队列中，实际上这个在RabbitMQ中不会发生。真实情况是，生产者将消息发送到**Exchange**（交换器，通常也可以用大写的“X”来表示），由交换器将消息路由到一个或者多个队列中。如果路由不到，或许会返回给生产者，或许直接丢弃。这里可以将RabbitMQ中的交换器看作一个简单的实体，更多的细节会在后面的章节中有所涉及。

交换器的具体示意图如图2-5所示。

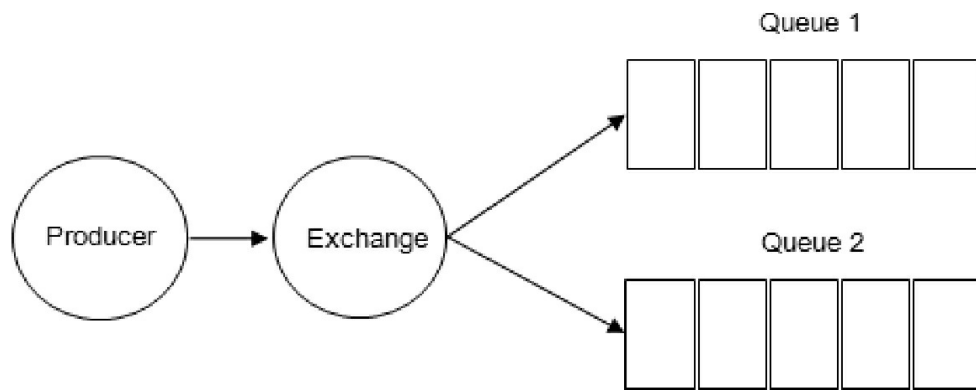


图2-5 交换器

RabbitMQ中的交换器有四种类型，不同的类型有着不同的路由策略，这将在下一节的交换器类型（Exchange Types）中介绍。

RoutingKey: 路由键。生产者将消息发给交换器的时候，一般会指定一个RoutingKey，用来指定这个消息的路由规则，而这个Routing Key需要与交换器类型和绑定键（BindingKey）联合使用才能最终生效。

在交换器类型和绑定键（BindingKey）固定的情况下，生产者可以在发送消息给交换器时，通过指定RoutingKey来决定消息流向哪里。

Binding: 绑定。RabbitMQ中通过绑定将交换器与队列关联起来，在绑定的时候一般会指定一个绑定键（**BindingKey**），这样RabbitMQ就知道如何正确地将消息路由到队列了，如图2-6所示。

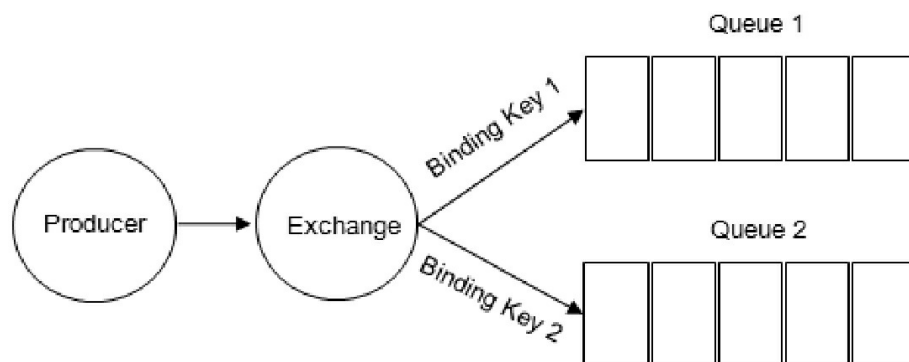


图2-6 绑定

生产者将消息发送给交换器时，需要一个**RoutingKey**，当**BindingKey**和**RoutingKey**相匹配时，消息会被路由到对应的队列中。在绑定多个队列到同一个交换器的时候，这些绑定允许使用相同的**BindingKey**。**BindingKey**并不是在所有的情况下都生效，它依赖于交换器类型，比如**fanout**类型的交换器就会无视**BindingKey**，而是将消息路由到所有绑定到该交换器的队列中。

对于初学者来说，交换器、路由键、绑定这几个概念理解起来会有点晦涩，可以对照着代码清单1-1来加深理解。

沿用本章开头的比喻，交换器相当于投递包裹的邮箱，**RoutingKey**相当于填写在包裹上的地址，**BindingKey**相当于包裹的目的地，当填写在包裹上的地址和实际想要投递的地址相匹配时，那么这个包裹就会被正确投递到目的地，最后这个目的地的“主人”——队列可以保留这个包裹。如果填写的地址出错，邮递员不能正确投递到目的地，包裹可能会回退给寄件人，也有可能被丢弃。

有经验的读者可能会发现，在某些情形下，**RoutingKey**与**BindingKey**可以看作同一个东西。代码清单2-1所展示的是代码清单1-1中的部分代码：

代码清单2-1 **RoutingKey**与**BindingKey**

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct", true, false, null);
channel.queueDeclare(QueueName, true, false, false, null);
channel.queueBind(QueueName, EXCHANGE_NAME, ROUTING_KEY);
String message = "Hello World!";
channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
    MessageProperties.PERSISTENT_TEXT_PLAIN,
    message.getBytes());
```

以上代码声明了一个direct类型的交换器（交换器的类型在下一节会详细讲述），然后将交换器和队列绑定起来。注意这里使用的字样是“ROUTING_KEY”，在本该使用BindingKey的channel.queueBind方法中却和channel.basicPublish方法同样使用了RoutingKey，这样做的潜台词是：这里的RoutingKey和BindingKey是同一个东西。在direct交换器类型下，RoutingKey和BindingKey需要完全匹配才能使用，所以上面代码中采用了此种写法会显得方便许多。

但是在topic交换器类型下，RoutingKey和BindingKey之间需要做模糊匹配，两者并不是相同的。

BindingKey其实也属于路由键中的一种，官方解释为：the routing key to use for the binding。可以翻译为：在绑定的时候使用的路由键。大多数时候，包括官方文档和RabbitMQ Java API中都把BindingKey和RoutingKey看作RoutingKey，为了避免混淆，可以这么理解：

✧ 在使用绑定的时候，其中需要的路由键是BindingKey。涉及的客户端方法如：channel.exchangeBind、channel.queueBind，对应的AMQP命令（详情参见2.2节）为Exchange.Bind、Queue.Bind。

✧ 在发送消息的时候，其中需要的路由键是RoutingKey。涉及的客户端方法如channel.basicPublish，对应的AMQP命令为Basic.Publish。

由于某些历史的原因，包括现存能搜集到的资料显示：大多数情况下习惯性地会将BindingKey写成RoutingKey，尤其是在使用direct类型的交换器的时候。本文后面的篇幅中也会将两者合称为路由键，读者需要注意区分其中的不同，可以根据上面的辨别方法进行有效的区分。

2.1.4 交换器类型

RabbitMQ常用的交换器类型有fanout、direct、topic、headers这四种。AMQP协议里还提到另外两种类型：System和自定义，这里不予描述。对于这四种类型下面一一阐述。

fanout

它会把所有发送到该交换器的消息路由到所有与该交换器绑定的队列中。

direct

direct类型的交换器路由规则也很简单，它会把消息路由到那些BindingKey和RoutingKey完全匹配的队列中。

以图2-7为例，交换器的类型为direct，如果我们发送一条消息，并在发送消息的时候设置路由键为“warning”，则消息会路由到Queue1和Queue2，对应的示例代码如下：

```
channel.basicPublish(EXCHANGE_NAME, "warning",  
    MessageProperties.PERSISTENT_TEXT_PLAIN,  
    message.getBytes());
```

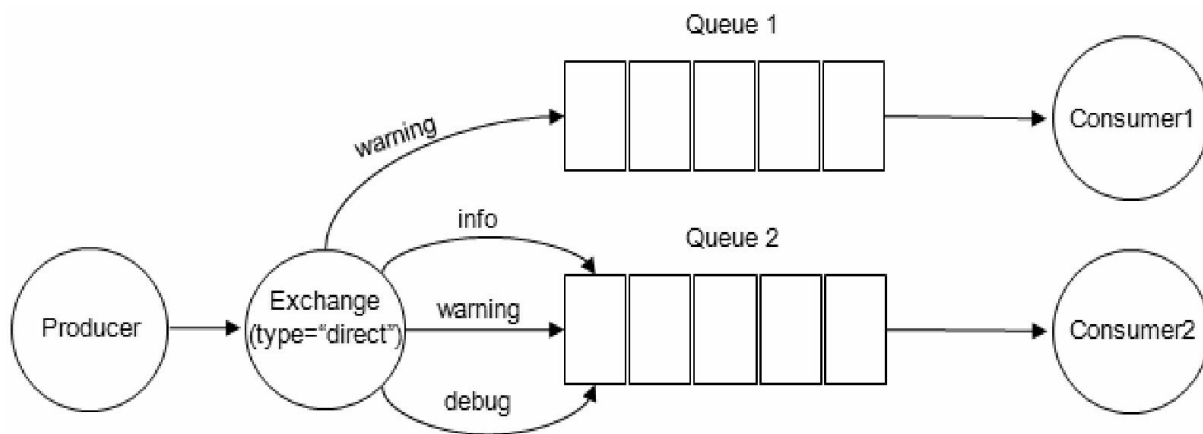


图2-7 direct类型的交换器

如果在发送消息的时候设置路由键为“info”或者“debug”，消息只会路由到Queue2。如果以其他的路由键发送消息，则消息不会路由到这两个队列中。

topic

前面讲到direct类型的交换器路由规则是完全匹配BindingKey和

RoutingKey，但是这种严格的匹配方式在很多情况下不能满足实际业务的需求。topic类型的交换器在匹配规则上进行了扩展，它与direct类型的交换器相似，也是将消息路由到BindingKey和RoutingKey相匹配的队列中，但这里的匹配规则有些不同，它约定：

✧ RoutingKey为一个点号“.”分隔的字符串（被点号“.”分隔开的每一段独立的字符串称为一个单词），如“com.rabbitmq.client”、“java.util.concurrent”、“com.hidden.client”；

✧ BindingKey和RoutingKey一样也是点号“.”分隔的字符串；

✧ BindingKey中可以存在两种特殊字符串“*”和“#”，用于做模糊匹配，其中“*”用于匹配一个单词，“#”用于匹配多规格单词（可以是零个）。

以图2-8中的配置为例：

- 路由键为“com.rabbitmq.client”的消息会同时路由到Queue1和Queue2；
- 路由键为“com.hidden.client”的消息只会路由到Queue2中；
- 路由键为“com.hidden.demo”的消息只会路由到Queue2中；
- 路由键为“java.rabbitmq.demo”的消息只会路由到Queue1中；
- 路由键为“java.util.concurrent”的消息将会被丢弃或者返回给生产者（需要设置mandatory参数），因为它没有匹配任何路由键。

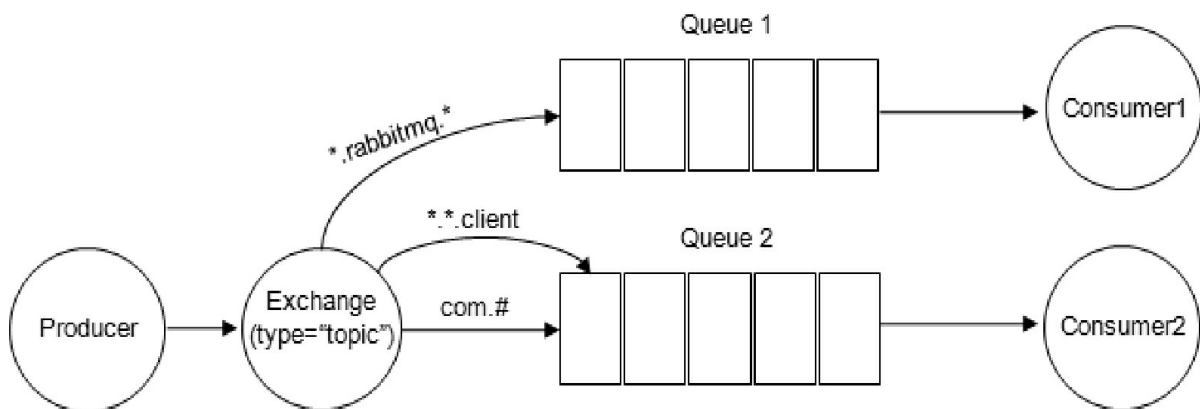


图2-8 topic类型的交换器

headers

headers类型的交换器不依赖于路由键的匹配规则来路由消息，而是根据发送的消息内容中的headers属性进行匹配。在绑定队列和交换器时制定一组键值对，当发送消息到交换器时，RabbitMQ会获取到该消息的headers（也是一个键值对的形式），对比其中的键值对是否完全匹配队列和交换器绑定时指定的键值对，如果完全匹配则消息会路由到该队列，否则不会路由到该队列。headers类型的交换器性能会很差，而且也不实用，基本上不会看到它的存在。

2.1.5 RabbitMQ运转流程

了解了以上的RabbitMQ架构模型及相关术语，再来回顾整个消息队列的使用过程。在最初状态下，生产者发送消息的时候（可依照图2-1）：

- （1）生产者连接到RabbitMQ Broker，建立一个连接（Connection），开启一个信道（Channel）（详细内容请参考3.1节）。
- （2）生产者声明一个交换器，并设置相关属性，比如交换机类型、是否持久化等（详细内容请参考3.2节）。
- （3）生产者声明一个队列并设置相关属性，比如是否排他、是否持久化、是否自动删除等（详细内容请参考3.2节）。
- （4）生产者通过路由键将交换器和队列绑定起来（详细内容请参考3.2节）。
- （5）生产者发送消息至RabbitMQ Broker，其中包含路由键、交换器等信息（详细内容请参考3.3节）。
- （6）相应的交换器根据接收到的路由键查找相匹配的队列。
- （7）如果找到，则将从生产者发送过来的消息存入相应的队列中。
- （8）如果没有找到，则根据生产者配置的属性选择丢弃还是回退给生产者（详细内容请参考4.1节）。
- （9）关闭信道。

(10) 关闭连接。

消费者接收消息的过程：

(1) 消费者连接到RabbitMQ Broker，建立一个连接（Connection），开启一个信道（Channel）。

(2) 消费者向RabbitMQ Broker请求消费相应队列中的消息，可能会设置相应的回调函数，以及做一些准备工作（详细内容请参考3.4节）。

(3) 等待RabbitMQ Broker回应并投递相应队列中的消息，消费者接收消息。

(4) 消费者确认（ack）接收到的消息。

(5) RabbitMQ从队列中删除相应已经被确认的消息。

(6) 关闭信道。

(7) 关闭连接。

如图2-9所示，我们又引入了两个新的概念：Connection和Channel。我们知道无论是生产者还是消费者，都需要和RabbitMQ Broker建立连接，这个连接就是一条TCP连接，也就是Connection。一旦TCP连接建立起来，客户端紧接着可以创建一个AMQP信道（Channel），每个信道都会被指派一个唯一的ID。信道是建立在Connection之上的虚拟连接，RabbitMQ处理的每条AMQP指令都是通过信道完成的。

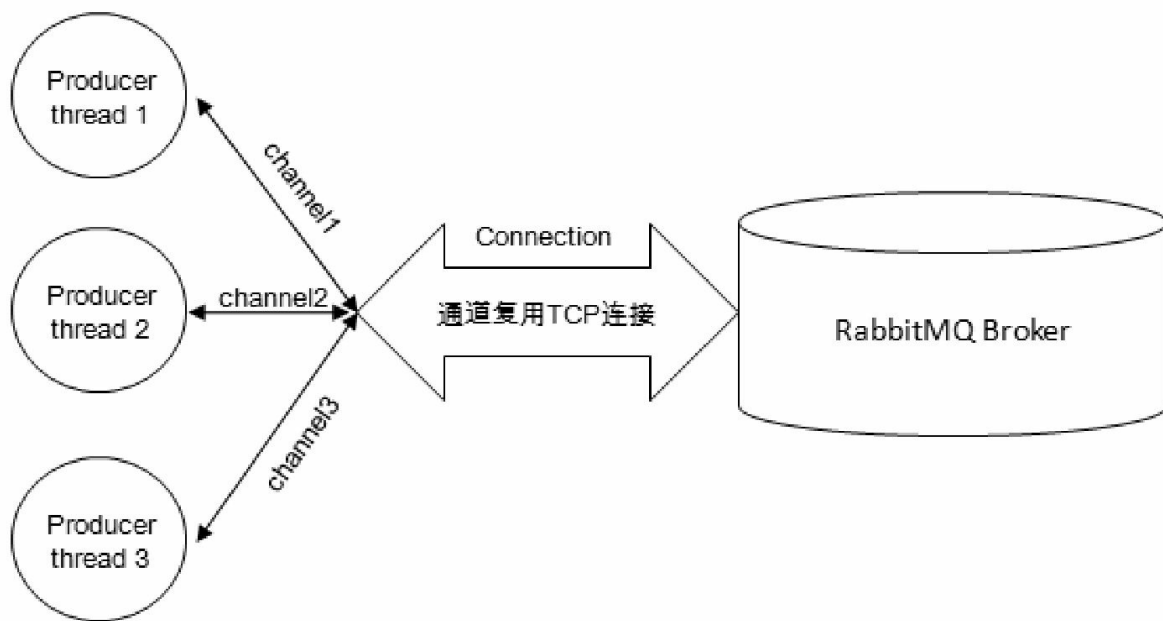


图2-9 Connection与Channel

我们完全可以直接使用Connection就能完成信道的工作，为什么还要引入信道呢？试想这样一个场景，一个应用程序中有很多个线程需要从RabbitMQ中消费消息，或者生产消息，那么必然需要建立很多个Connection，也就是许多个TCP连接。然而对于操作系统而言，建立和销毁TCP连接是非常昂贵的开销，如果遇到使用高峰，性能瓶颈也随之显现。RabbitMQ采用类似NIO^[1]（Non-blocking I/O）的做法，选择TCP连接复用，不仅可以减少性能开销，同时也便于管理。

每个线程把持一个信道，所以信道复用了Connection的TCP连接。同时RabbitMQ可以确保每个线程的私密性，就像拥有独立的连接一样。当每个信道的流量不是很大时，复用单一的Connection可以在产生性能瓶颈的情况下有效地节省TCP连接资源。但是当信道本身的流量很大时，这时候多个信道复用一个Connection就会产生性能瓶颈，进而使整体的流量被限制了。此时就需要开辟多个Connection，将这些信道均摊到这些Connection中，至于这些相关的调优策略需要根据业务自身的实际情况进行调节，更多内容可以参考第9章。

信道在AMQP中是一个很重要的概念，大多数操作都是在信道这个层面展开的。在代码清单1-1中也可以看出一些端倪，比如

`channel.exchangeDeclare`、`channel.queueDeclare`、`channel.basicPublish`和`channel.basicConsume`等方法。RabbitMQ相关的API与AMQP紧密相连，比如`channel.basicPublish`对应AMQP的`Basic.Publish`命令，在下面的小节中将会为大家一一展开。

2.2 AMQP协议介绍

从前面的内容可以了解到RabbitMQ是遵从AMQP协议的，换句话说，RabbitMQ就是AMQP协议的Erlang的实现（当然RabbitMQ还支持STOMP^[2]、MQTT^[3]等协议）。AMQP的模型架构和RabbitMQ的模型架构是一样的，生产者将消息发送给交换器，交换器和队列绑定。当生产者发送消息时所携带的RoutingKey与绑定时的BindingKey相匹配时，消息即被存入相应的队列之中。消费者可以订阅相应的队列来获取消息。

RabbitMQ中的交换器、交换器类型、队列、绑定、路由键等都是遵循的AMQP协议中相应的概念。目前RabbitMQ最新版本默认支持的是AMQP 0-9-1。本书中如无特殊说明，都以AMQP 0-9-1为基准进行介绍。

AMQP协议本身包括三层：

✧ **Module Layer:** 位于协议最高层，主要定义了一些供客户端调用的命令，客户端可以利用这些命令实现自己的业务逻辑。例如，客户端可以使用Queue.Declare命令声明一个队列或者使用Basic.Consume订阅消费一个队列中的消息。

✧ **Session Layer:** 位于中间层，主要负责将客户端的命令发送给服务器，再将服务端的应答返回给客户端，主要为客户端与服务器之间的通信提供可靠性同步机制和错误处理。

✧ **Transport Layer:** 位于最底层，主要传输二进制数据流，提供帧的处理、信道复用、错误检测和数据表示等。

AMQP说到底还是一个通信协议，通信协议都会涉及报文交互，从low-level层面举例来说，AMQP本身是应用层的协议，其填充于TCP协议层的数据部分。而从high-level层面来说，AMQP是通过协议命令进行交互的。AMQP协议可以看作一系列结构化命令的集合，这里的命令代表一种操作，类似于HTTP中的方法（GET、POST、PUT、DELETE等）。

2.2.1 AMQP生产者流转过程

为了形象地说明AMQP协议命令的流转过程，这里截取代码清单1-1中的关键代码，如代码清单2-2所示。

代码清单2-2 简洁版生产者代码

```
Connection connection = factory.newConnection();//创建连接
Channel channel = connection.createChannel();//创建信道
String message = "Hello World!";
channel.basicPublish(EXCHANGE_NAME, ROUTING_KEY,
    MessageProperties.PERSISTENT_TEXT_PLAIN,
    message.getBytes());
//关闭资源
channel.close();
connection.close();
```

当客户端与Broker建立连接的时候，会调用factory.newConnection方法，这个方法会进一步封装成Protocol Header 0-9-1的报文头发送给Broker，以此通知Broker本次交互采用的是AMQP 0-9-1协议，紧接着Broker返回Connection.Start来建立连接，在连接的过程中涉及Connection.Start/.Start-OK、Connection.Tune/.Tune-Ok、Connection.Open/.Open-Ok这6个命令的交互。

当客户端调用connection.createChannel方法准备开启信道的时候，其包装Channel.Open命令发送给Broker，等待Channel.Open-Ok命令。

当客户端发送消息的时候，需要调用channel.basicPublish方法，对应的AMQP命令为Basic.Publish，注意这个命令和前面涉及的命令略有不同，这个命令还包含了Content Header和Content Body。Content Header里面包含的是消息体的属性，例如，投递模式（可以参考3.3节）、优先级等，而ContentBody包含消息体本身。

当客户端发送完消息需要关闭资源时，涉及Channel.Close/.CClose-Ok与Connection.Close/.Close-Ok的命令交互。详细流转过程如图2-10所示。

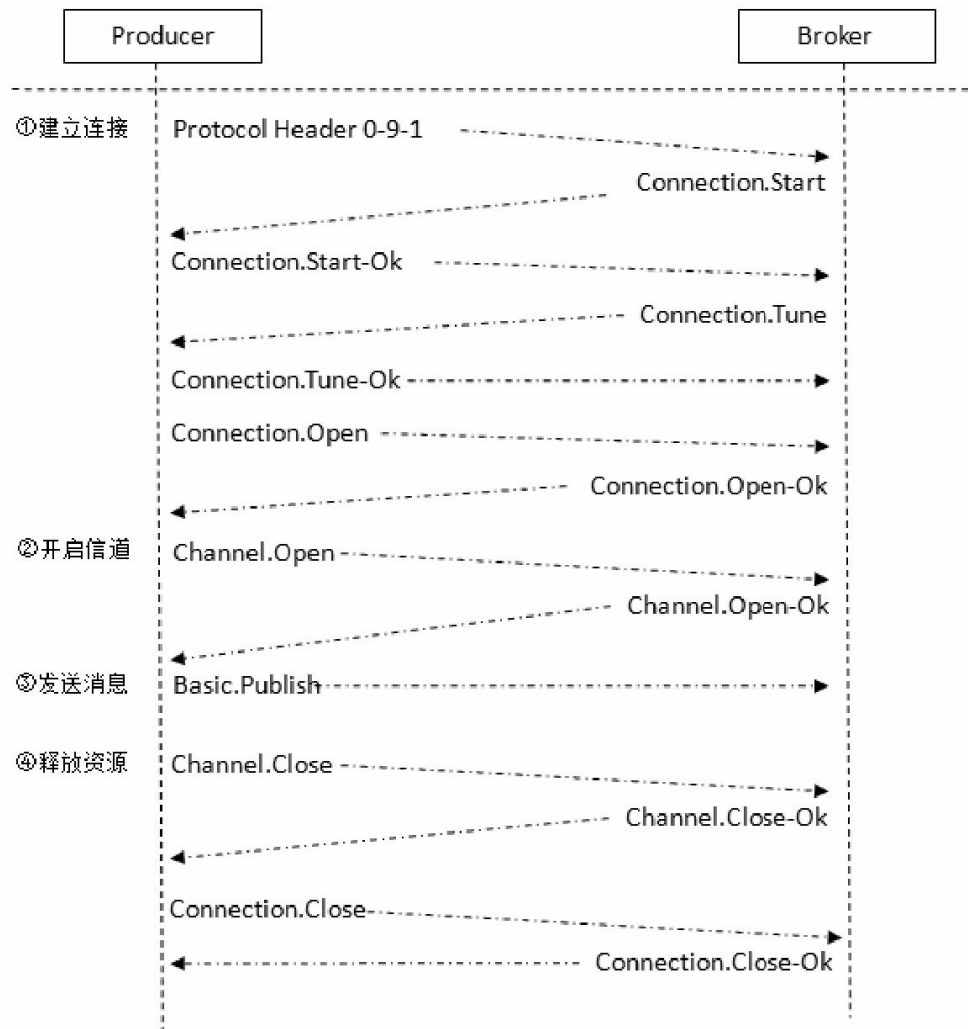


图2-10 流转过程

22.2.2 AMQPP 消费者流转过程

本节我们继续来看消费者的流转过程，参考代码清单1-2，截取消费端的关键代码如代码清单2-3所示。

代码清单2-3 简洁版消费者代码

```
Connection cconnection = factory.newConnnection (addresses) ;//创建连接
```

```
final Channeel channel = connection.creaateChannel ( ) ;///创建信道
```

```
Consumer connsumer = new DefaultConsumerr (channel) {}///.....
```

省略实现

```
channel.basicQos (64) ;  
channel.basicConsume (QUEUE_NAME,consumer) ;  
//等待回调函数执行完毕之后，关闭资源  
TimeUnit.SECONDS.sleep (5) ;  
channel.close () ;  
connection.close () ;
```

其详细流转过程如图2-11所示。

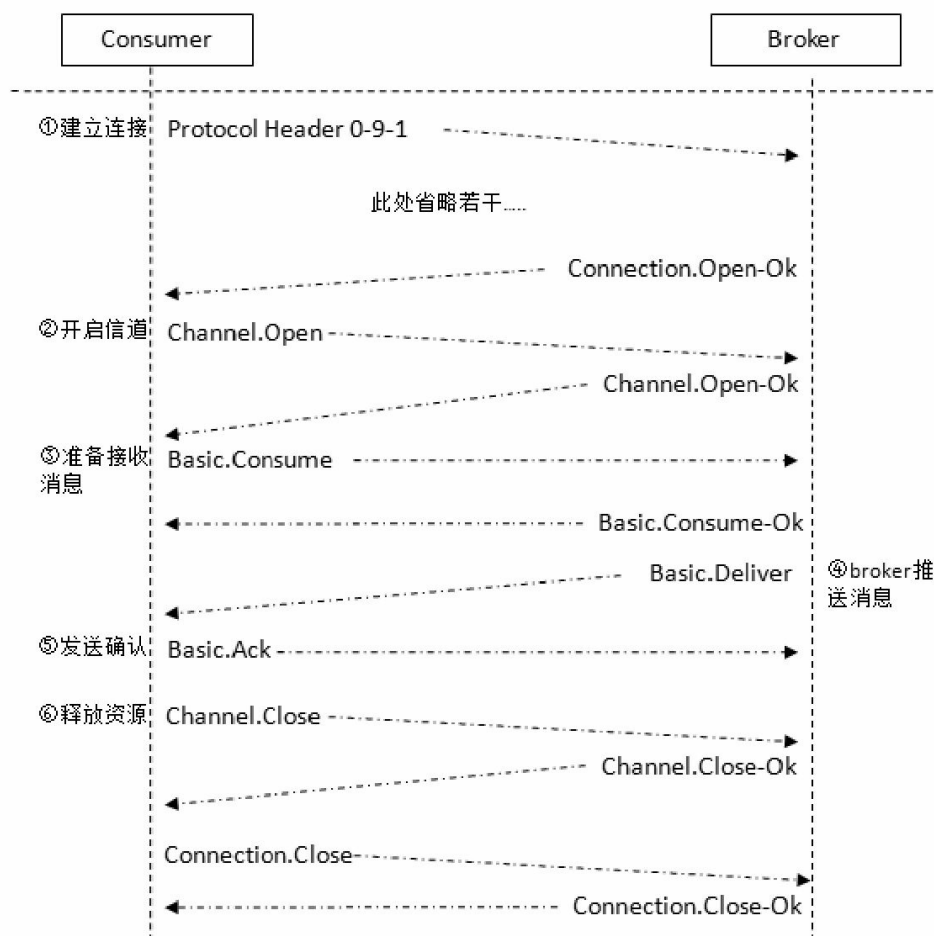


图2-11 流转过程

消费者客户端同样需要与Broker建立连接，与生产者客户端一样，

协议交互同样涉及Connection.Start/.Start-Ok、Connection.Tune/.Tune-Ok和Connection. Open/.Open-Ok等，图2-11中省略了这些步骤，可以参考图2-10。

紧接着也少不了在Connection之上建立Channel，和生产者客户端一样，协议涉及Channel.Open/Open-Ok。

如果在消费之前调用了channel.basicQos（int prefetchCount）的方法来设置消费者客户端最大能“保持”的未确认的消息数（即预取个数），那么协议流转会涉及Basic.Qos/.Qos-Ok这两个AMQP命令。

在真正消费之前，消费者客户端需要向Broker发送Basic.Consume命令（即调用channel.basicConsume方法）将Channel置为接收模式，之后Broker回执Basic.Consume-Ok以告诉消费者客户端准备好消费消息。紧接着Broker向消费者客户端推送（Push）消息，即Basic.Deliver命令，有意思的是这个和Basic.Publish命令一样会携带Content Header和Content Body。

消费者接收到消息并正确消费之后，向Broker发送确认，即Basic.Ack命令。

在消费者停止消费的时候，主动关闭连接，这点和生产者一样，涉及Channel.Close/.Close-Ok和Connection.Close/.Close-Ok。

2.2.3 AMQP命令概览

AMQP 0-9-1协议中的命令远远不止上面所涉及的这些，为了让读者在遇到其他命令的时候能够迅速查阅相关信息，下面列举了AMQP 0-9-1协议主要的命令，包含名称、是否包含内容体（Content Body）、对应客户端中相应的方法及简要描述等四个维度进行说明，具体如表2-1所示。

表2-1 AMQP命令

名 称	是否包含内容体	对应客户端中的方法	简要描述
Connection.Start	否	factory.newConnection	建立连接相关
Connection.Start-Ok	否	同上	同上

续表

名 称	是否包含内容体	对应客户端中的方法	简要描述
Connection.Tune	否	同上	同上
Connection.Tune-Ok	否	同上	同上
Connection.Open	否	同上	同上
Connection.Open-Ok	否	同上	同上
Connection.Close	否	connection.close	关闭连接
Connection.Close-Ok	否	同上	同上
Channel.Open	否	connection.openChannel	开启信道
Channel.Open-Ok	否	同上	同上
Channel.Close	否	channel.close	关闭信道
Channel.Close-Ok	否	同上	同上
Exchange.Declare	否	channel.exchangeDeclare	声明交换器
Exchange.Declare-Ok	否	同上	同上
Exchange.Delete	否	channel.exchangeDelete	删除交换器
Exchange.Delete-Ok	否	同上	同上
Exchange.Bind	否	channel.exchangeBind	交换器与交换器绑定
Exchange.Bind-Ok	否	同上	同上
Exchange.Unbind	否	channel.exchangeUnbind	交换器与交换器解绑
Exchange.Unbind-Ok	否	同上	同上
Queue.Declare	否	channel.queueDeclare	声明队列
Queue.Declare-Ok	否	同上	同上
Queue.Bind	否	channel.queueBind	队列与交换器绑定
Queue.Bind-Ok	否	同上	同上
Queue.Purge	否	channel.queuePurge	清除队列中的内容
Queue.Purge-Ok	否	同上	同上
Queue.Delete	否	channel.queueDelete	删除队列
Queue.Delete-Ok	否	同上	同上
Queue.Unbind	否	channel.queueUnbind	队列与交换器解绑
Queue.Unbind-Ok	否	同上	同上
Basic.Qos	否	channel.basicQos	设置未被确认消费的个数
Basic.Qos-Ok	否	同上	同上
Basic.Consume	否	channel.basicConsume	消费消息（推模式）
Basic.Consume-Ok	否	同上	同上
Basic.Cancel	否	channel.basicCancel	取消

续表

名 称	是否包含内容体	对应客户端中的方法	简要描述
Basic.Cancel-Ok	否	同上	同上
Basic.Publish	是	channel.basicPublish	发送消息
Basic.Return	是	无	未能成功路由的消息返回
Basic.Deliver	是	无	Broker 推送消息
Basic.Get	否	channel.basicGet	消费消息（拉模式）
Basic.Get-Ok	是	同上	同上
Basic.Ack	否	channel.basicAck	确认
Basic.Reject	否	channel.basicReject	拒绝（单条拒绝）
Basic.Recover	否	channel.basicRecover	请求 Broker 重新发送未被确认的消息
Basic.Recover-Ok	否	同上	同上
Basic.Nack	否	channel.basicNack	拒绝（可批量拒绝）
Tx.Select	否	channel.txSelect	开启事务
Tx.Select-Ok	否	同上	同上
Tx.Commit	否	channel.txCommit	事务提交
Tx.Commit-Ok	否	同上	同上
Tx.Rollback	否	channel.txRollback	事务回滚
Tx.Rollback-Ok	否	同上	同上
Confirm Select	否	channel.confirmSelect	开启发送端确认模式
Confirm.Select-Ok	否	同上	同上

2.3 小结

本章主要讲述的是RabbitMQ的入门知识，首先介绍了生产者（Producer）、消费者（Consumer）、队列（Queue）、交换器（Exchange）、路由键（RoutingKey）、绑定（Binding）、连接（Connection）和信道（Channel）等基本术语，还介绍了交换器的类型：fanout、direct、topic 和headers。之后通过介绍RabbitMQ的运转流程来加深对基本术语的理解。

RabbitMQ可以看作AMQP协议的具体实现，2.2节还大致介绍了AMQP命令以及与RabbitMQ客户端中方法如何一一对应，包括对各个整个生产消费消息的AMQP命令的流程介绍。最后展示了AMQP 0-9-1中常用的命令与RabbitMQ客户端中方法的映射关系。

[1] NIO，也称非阻塞I/O，包含三大核心部分：Channel（信道）、Buffer（缓冲区）和Selector（选择器）。NIO基于Channel和Buffer进行操作，数据总是从信道读取数据到缓冲区中，或者从缓冲区写入到信道中。Selector用于监听多个信道的事件（比如连接打开，数据到达等）。因此，单线程可以监听多个数据的信道。NIO中有一个很有名的Reactor模式，有兴趣的读者可以深入研究。

[2] STOMP，即Simple（or Streaming）Text Oriented Messaging Protocol，简单（流）文本面向消息协议，它提供了一个可互操作的连接格式，运行STOMP客户端与任意STOMP消息代理（Broker）进行交互。STOMP协议由于设计简单，易于开发客户端，因此在多种语言和平台上得到广泛的应用。

[3] MQTT，即Message Queuing Telemetry Transport，消息队列遥测传输，是IBM开发的一个即时通信协议，有可能成为物联网的重要组成部分。该协议支持所有平台，几乎可以把所有物联网和外部连接起来，被用来当作传感器和制动器的通信协议。

第3章 客户端开发向导

RabbitMQ Java客户端使用`com.rabbitmq.client`作为顶级包名，关键的Class和Interface有Channel、Connection、ConnectionFactory、Consumer等。AMQP协议层面的操作通过Channel接口实现。Connection是用来开启Channel（信道）的，可以注册事件处理器，也可以在应用结束时关闭连接。与RabbitMQ相关的开发工作，基本上也是围绕Connection和Channel这两个类展开的。本章按照一个完整的运转流程进行讲解，详细内容有几点：连接、交换器/队列的创建与绑定、发送消息、消费消息、消费消息的确认和关闭连接。

3.1 连接RabbitMQ

下面的代码（代码清单3-1）用来在给定的参数（IP地址、端口号、用户名、密码等）下连接RabbitMQ：

代码清单 3-1

```
ConnectionFactory factory = new ConnectionFactory ();  
factory.setUsername (USERNAME);  
factory.setPassword (PASSWORD);  
factory.setVirtualHost (virtualHost);  
factory.setHost (IP_ADDRESS);  
factory.setPort (PORT);
```

```
Connection conn = factory.newConnection ();
```

也可以选择使用URI的方式来实现，示例如代码清单3-2所示。

代码清单 3-2

```
ConnectionFactory factory = new ConnectionFactory ();  
factory.setUri ("amqp://userName:password@ipAddress:portNumber/vi  
Connection conn = factory.newConnection ();
```

Connection接口被用来创建一个Channel：

```
Channel channel = conn.createChannel ();
```

在创建之后，Channel可以用来发送或者接收消息了。

注意要点：

Connection可以用来创建多个Channel实例，但是Channel实例不能在线程间共享，应用程序应该为每一个线程开辟一个Channel。某些情况下Channel的操作可以并发运行，但是在其他情况下会导致在网络上出现错误的通信帧交错，同时也会影响发送方确认（publisher confirm）

机制的运行（详细可以参考4.8节），所以多线程间共享Channel实例是非线程安全的。

Channel或者Connection中有个isOpen方法可以用来检测其是否已处于开启状态（关于Channel或者Connection的状态可以参考3.6节）。但并不推荐在生产环境的代码上使用isOpen方法，这个方法的返回值依赖于shutdownCause（参考下面的代码）的存在，有可能会产生竞争，代码清单3-3是isOpen方法的源码：

代码清单3-3 isOpen方法的源码

```
public boolean isOpen() {
    synchronized(this.monitor) {
        return this.shutdownCause == null;
    }
}
```

错误地使用isOpen方法示例代码如代码清单3-4所示。

代码清单3-4 错误地使用isOpen方法

```
public void brokenMethod(Channel channel)
{
    if (channel.isOpen())
    {
        // The following code depends on the channel being in open state.
        // However there is a possibility of the change in the channel state
        // between isOpen() and basicQos(1) call
        ...
        channel.basicQos(1);
    }
}
```

通常情况下，在调用createXXX或者newXXX方法之后，我们可以简单地认为Connection或者Channel已经成功地处于开启状态，而并不会在代码中使用isOpen这个检测方法。如果在使用Channel的时候其已经处于关闭状态，那么程序会抛出一个com.rabbitmq.client.ShutdownSignalException，我们只需捕获这个异常即可。当然同时也要试着捕获IOException或者SocketException，以防Connection意外关闭。示例代码如代码清单3-5所示。

代码清单3-5

```
public void validMethod(Channel channel)
{
    try {
        ...
        channel.basicQos(1);
    } catch (ShutdownSignalException sse) {
        // possibly check if channel was closed
        // by the time we started action and reasons for
        // closing it
        ...

        } catch (IOException ioe) {
            // check why connection was closed
            ...
        }
    }
}
```

3.2 使用交换器和队列

交换器和队列是AMQP中high-level层面的构建模块，应用程序需确保在使用它们的时候就已经存在了，在使用之前需要先声明（declare）它们。

代码清单3-6演示了如何声明一个交换器和队列：

代码清单 3-6

```
channel.exchangeDeclare (exchangeName, "direct", true) ;  
String queueName = channel.queueDeclare ( ) .getQueue ( ) ;  
channel.queueBind (queueName, exchangeName, routingKey) ;
```

上面创建了一个持久化的、非自动删除的、绑定类型为direct的交换器，同时也创建了一个非持久化的、排他的、自动删除的队列（此队列的名称由RabbitMQ自动生成）。这里的交换器和队列也都没有设置特殊的参数。

上面的代码也展示了如何使用路由键将队列和交换器绑定起来。上面声明的队列具备如下特性：只对当前应用中同一个Connection层面可用，同一个Connection的不同Channel可共用，并且也会在应用连接断开时自动删除。

如果要在应用中共享一个队列，可以做如下声明，如代码清单3-7所示。

代码清单 3-7

```
channel.exchangeDeclare (exchangeName, "direct", true) ;  
channel.queueDeclare (queueName, true, false, false, null) ;  
channel.queueBind (queueName, exchangeName, routingKey) ;
```

这里的队列被声明为持久化的、非排他的、非自动删除的，而且也被分配另一个确定的已知的名称（由客户端分配而非RabbitMQ自动生成）。

注意：Channel的API方法都是可以重载的，比如exchangeDeclare、queueDeclare。根据参数不同，可以有不同的重载形式，根据自身的需要进行调用。

生产者和消费者都可以声明一个交换器或者队列。如果尝试声明一个已经存在的交换器或者队列，只要声明的参数完全匹配现存的交换器或者队列，RabbitMQ就可以什么都不做，并成功返回。如果声明的参数不匹配则会抛出异常。

3.2.1 exchangeDeclare方法详解

exchangeDeclare有多个重载方法，这些重载方法都是由下面这个方法中缺省的某些参数构成的。

```
Exchange.DeclareOk exchangeDeclare(String exchange,
                                     String type, boolean durable,
                                     boolean autoDelete, boolean internal,
                                     Map<String, Object> arguments) throws IOException;
```

这个方法的返回值是Exchange.DeclareOK，用来标识成功声明了一个交换器。

各个参数详细说明如下所述。

- ✧ **exchange**：交换器的名称。
- ✧ **type**：交换器的类型，常见的如fanout、direct、topic，详情参见2.1.4节。
- ✧ **durable**：设置是否持久化。durable设置为true表示持久化，反之是非持久化。持久化可以将交换器存盘，在服务器重启的时候不会丢失相关信息。
- ✧ **autoDelete**：设置是否自动删除。autoDelete设置为true则表示自动删除。自动删除的前提是至少有一个队列或者交换器与这个交换器绑定，之后所有与这个交换器绑定的队列或者交换器都与此解绑。注意不能错误地把这个参数理解为：“当与此交换器连接的客户端都断开时，RabbitMQ会自动删除本交换器”。

✧ **internal**: 设置是否是内置的。如果设置为true, 则表示是内置的交换器, 客户端程序无法直接发送消息到这个交换器中, 只能通过交换器路由到交换器这种方式。

✧ **argument**: 其他一些结构化参数, 比如alternate-exchange (有关alternateexchange的详情可以参考4.1.3节)。

exchangeDeclare的其他重载方法如下:

(1) Exchange.DeclareOk exchangeDeclare (String exchange, String type) throws IOException;

(2) Exchange.DeclareOk exchangeDeclare (String exchange, String type, boolean durable) throws IOException;

(3) Exchange.DeclareOk exchangeDeclare (String exchange, String type, boolean durable, boolean autoDelete, Map<String, Object> arguments) throws IOException;

与此对应的, 将第二个参数String type换成BuiltInExchangeType type对应的几个重载方法 (不常用):

(1) Exchange.DeclareOk exchangeDeclare (String exchange, BuiltInExchangeType type) throws IOException;

(2) Exchange.DeclareOk exchangeDeclare (String exchange, BuiltInExchangeType type, boolean durable) throws IOException;

(3) Exchange.DeclareOk exchangeDeclare (String exchange, BuiltInExchangeType type, boolean durable, boolean autoDelete, Map<String, Object> arguments) throws IOException;

(4) Exchange.DeclareOk exchangeDeclare (String exchange, BuiltInExchangeType type, boolean durable, boolean autoDelete, boolean internal, Map<String, Object> arguments) throws IOException;

与exchangeDeclare师出同门的还有几个方法, 比如exchangeDeclareNoWait方法, 具体定义如下 (当然也有BuiltExchangeType版的, 这里就不展开了):

```
void exchangeDeclareNoWait(String exchange,
                            String type,
                            boolean durable,
                            boolean autoDelete,
                            boolean internal,
                            Map<String, Object> arguments) throws IOException;
```

这个exchangeDeclareNoWait比exchangeDeclare多设置了一个nowait参数，这个nowait参数指的是AMQP中Exchange.Declare命令的参数，意思是不需要服务器返回，注意这个方法的返回值是void，而普通的exchangeDeclare方法的返回值是Exchange.DeclareOk，意思是在客户端声明了一个交换器之后，需要等待服务器的返回（服务器会返回Exchange.Declare-Ok这个AMQP命令）。

针对“exchangeDeclareNoWait不需要服务器任何返回值”这一点，考虑这样一种情况，在声明完一个交换器之后（实际服务器还并未完成交换器的创建），那么此时客户端紧接着使用这个交换器，必然会发生异常。如果没有特殊的缘由和应用场景，并不建议使用这个方法。

这里还有师出同门的另一个方法exchangeDeclarePassive，这个方法定义如下：

```
Exchange.DeclareOk exchangeDeclarePassive (String name) throws
IOException;
```

这个方法在实际应用过程中还是非常有用的，它主要用来检测相应的交换器是否存在。如果存在则正常返回；如果不存在则抛出异常：404 channel exception，同时Channel也会被关闭。

有声明创建交换器的方法，当然也有删除交换器的方法。相应的方法如下：

```
(1) Exchange.DeleteOk exchangeDelete (String exchange) throws
IOException;
```

```
(2) void exchangeDeleteNoWait (String exchange, boolean
ifUnused) throws IOException;
```

```
(3) Exchange.DeleteOk exchangeDelete (String exchange, boolean
ifUnused) throws IOException;
```

其中exchange表示交换器的名称，而ifUnused用来设置是否在交换器没有被使用的情况下删除。如果isUnused设置为true，则只有在此交换器没有被使用的情况下才会被删除；如果设置false，则无论如何这个交换器都要被删除。

3.2.2 queueDeclare方法详解

queueDeclare相对于exchangeDeclare方法而言，重载方法的个数就少很多，它只有两个重载方法：

(1) Queue.DeclareOk queueDeclare () throws IOException;

(2) Queue.DeclareOk queueDeclare (String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments) throws IOException;

不带任何参数的queueDeclare方法默认创建一个由RabbitMQ命名的（类似这种amq.gen-LhQz1gv3GhDOv8PIDabOXA名称，这种队列也称为匿名队列）、排他的、自动删除的、非持久化的队列。

方法的参数详细说明如下所述。

✧ queue：队列的名称。

✧ durable：设置是否持久化。为true则设置队列为持久化。持久化的队列会存盘，在服务器重启的时候可以保证不丢失相关信息。

✧ exclusive：设置是否排他。为true则设置队列为排他的。如果一个队列被声明为排他队列，该队列仅对首次声明它的连接可见，并在连接断开时自动删除。这里需要注意三点：排他队列是基于连接（Connection）可见的，同一个连接的不同信道（Channel）是可以同时访问同一连接创建的排他队列；“首次”是指如果一个连接已经声明了一个排他队列，其他连接是不允许建立同名的排他队列的，这个与普通队列不同；即使该队列是持久化的，一旦连接关闭或者客户端退出，该排他队列都会被自动删除，这种队列适用于一个客户端同时发送和读取消息的应用场景。

✧ autoDelete：设置是否自动删除。为true则设置队列为自动删

除。自动删除的前提是：至少有一个消费者连接到这个队列，之后所有与这个队列连接的消费者都断开时，才会自动删除。不能把这个参数错误地理解为：“当连接到此队列的所有客户端断开时，这个队列自动删除”，因为生产者客户端创建这个队列，或者没有消费者客户端与这个队列连接时，都不会自动删除这个队列。

✧ arguments: 设置队列的其他一些参数，如x-message-ttl、x-expires、x-max-length、x-max-length-bytes、x-dead-letter-exchange、x-deadletter-routing-key、x-max-priority等。

注意要点：

生产者和消费者都能够使用queueDeclare来声明一个队列，但是如果消费者在同一个信道上订阅了另一个队列，就无法再声明队列了。必须先取消订阅，然后将信道置为“传输”模式，之后才能声明队列。

对应于exchangeDeclareNoWait方法，这里也有一个queueDeclareNoWait方法：

```
void queueDeclareNoWait (String queue, boolean durable, boolean exclusive, boolean autoDelete, Map<String, Object> arguments) throws IOException;
```

方法的返回值也是void，表示不需要服务端的任何返回。同样也需要注意，在调用完queueDeclareNoWait方法之后，紧接着使用声明的队列时有可能会发生异常情况。

同样这里还有一个queueDeclarePassive的方法，也比较常用。这个方法用来检测相应的队列是否存在。如果存在则正常返回，如果不存在则抛出异常：404 channel exception，同时Channel也会被关闭。方法定义如下：

```
Queue.DeclareOk queueDeclarePassive (String queue) throws IOException;
```

与交换器对应，关于队列也有删除的相应方法：

```
(1) Queue.DeleteOk queueDelete (String queue) throws IOException;
```

```
(2) Queue.DeleteOk queueDelete (String queue, boolean ifUnused,
```

boolean isEmpty) throws IOException;

(3) void queueDeleteNoWait (String queue, boolean ifUnused, boolean isEmpty) throws IOException;

其中queue表示队列的名称，ifUnused可以参考上一小节的交换器。isEmpty设置为true表示在队列为空（队列里面没有任何消息堆积）的情况下才能够删除。

与队列相关的还有一个有意思的方法——queuePurge，区别于queueDelete，这个方法用来清空队列中的内容，而不删除队列本身，具体定义如下：

Queue.PurgeOk queuePurge (String queue) throws IOException;

3.2.3 queueBind方法详解

将队列和交换器绑定的方法如下，可以与前两节中的方法定义进行类比。

(1) Queue.BindOk queueBind (String queue, String exchange, String routingKey) throws IOException;

(2) Queue.BindOk queueBind (String queue, String exchange, String routingKey, Map<String, Object> arguments) throws IOException;

(3) void queueBindNoWait (String queue, String exchange, String routingKey, Map<String, Object> arguments) throws IOException;

方法中涉及的参数详解。

- ✧ queue: 队列名称;
- ✧ exchange: 交换器的名称;
- ✧ routingKey: 用来绑定队列和交换器的路由键;
- ✧ argument: 定义绑定的一些参数。

不仅可以将队列和交换器绑定起来，也可以将已经被绑定的队列和交换器进行解绑。具体方法可以参考如下（具体的参数解释可以参考前

面的内容，这里不再赘述）：

（1） Queue.UnbindOk queueUnbind (String queue, String exchange, String routingKey) throws IOException;

（2） Queue.UnbindOk queueUnbind (String queue, String exchange, String routingKey, Map<String, Object> arguments) throws IOException;

3.2.4 exchangeBind方法详解

我们不仅可以将交换器与队列绑定，也可以将交换器与交换器绑定，后者和前者的用法如出一辙，相应的方法如下：

（1） Exchange.BindOk exchangeBind (String destination, String source, String routingKey) throws IOException;

（2） Exchange.BindOk exchangeBind (String destination, String source, String routingKey, Map<String, Object> arguments) throws IOException;

（3） void exchangeBindNoWait (String destination, String source, String routingKey, Map<String, Object> arguments) throws IOException;

方法中的参数可以参考3.2.1节的exchangeDeclare方法。绑定之后，消息从source交换器转发到destination交换器，某种程度上来说destination交换器可以看作一个队列。示例代码如代码清单3-8所示。

代码清单3-8

```
channel.exchangeDeclare ("source", "direct", false, true, null) ;
channel.exchangeDeclare ("destination", "fanout", false, true, null) ;
channel.exchangeBind ("destination", "source", "exKey") ;
channel.queueDeclare ("queue", false, false, true, null) ;
channel.queueBind ("queue", "destination", "") ;
channel.basicPublish ("source", "exKey", null,
"exToExDemo".getBytes ( ) ) ;
```

生产者发送消息至交换器source中，交换器source根据路由键找到与其匹配的另一个交换器destination，并把消息转发到destination中，进而存储在destination绑定的队列queue中，可参考图3-1。

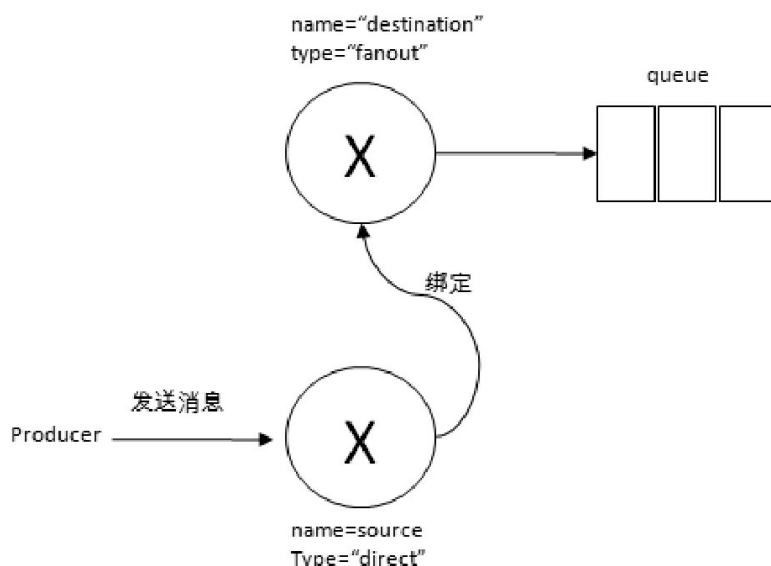


图3-1 交换器与交换器绑定

3.2.5 何时创建

RabbitMQ的消息存储在队列中，交换器的使用并不真正耗费服务器的性能，而队列会。如果要衡量RabbitMQ当前的QPS^[4]只需看队列的即可。在实际业务应用中，需要对所创建的队列的流量、内存占用及网卡占用有一个清晰的认知，预估其平均值和峰值，以便在固定硬件资源的情况下能够进行合理有效的分配。

按照RabbitMQ官方建议，生产者和消费者都应该尝试创建（这里指声明操作）队列。这是一个很好的建议，但不适用于所有的情况。如果业务本身在架构设计之初已经充分地预估了队列的使用情况，完全可以在业务程序上线之前在服务器上创建好（比如通过页面管理、RabbitMQ命令或者更好的是从配置中心下发），这样业务程序也可以免去声明的过程，直接使用即可。

预先创建好资源还有一个好处是，可以确保交换器和队列之间正确地绑定匹配。很多时候，由于人为因素、代码缺陷等，发送消息的交换

器并没有绑定任何队列，那么消息将会丢失；或者交换器绑定了某个队列，但是发送消息时的路由键无法与现存的队列匹配，那么消息也会丢失。当然可以配合mandatory参数或者备份交换器（详细可参考4.1节）来提高程序的健壮性。

与此同时，预估好队列的使用情况非常重要，如果在后期运行过程中超过预定的阈值，可以根据实际情况对当前集群进行扩容或者将相应的队列迁移到其他集群。迁移的过程也可以对业务程序完全透明。此种方法也更有利于开发和运维分工，便于相应资源的管理。

如果集群资源充足，而即将使用的队列所占用的资源又在可控的范围之内，为了增加业务程序的灵活性，也完全可以在业务程序中声明队列。

至于是使用预先分配创建资源的静态方式还是动态的创建方式，需要从业务逻辑本身、公司运维体系和公司硬件资源等方面考虑。

3.3 发送消息

如果要发送一个消息，可以使用Channel类的basicPublish方法，比如发送一条内容为“Hello World!”的消息，参考如下：

```
byte[] messageBodyBytes = "Hello, world!".getBytes ( ) ;  
channel.basicPublish (exchangeName,          routingKey,          null,  
messageBodyBytes) ;
```

为了更好地控制发送，可以使用mandatory这个参数，或者可以发送一些特定属性的信息：

```
channel.basicPublish(exchangeName, routingKey, mandatory,  
                     MessageProperties.PERSISTENT_TEXT_PLAIN,  
                     messageBodyBytes) ;
```

免费样章到此结束。

喜欢这本书？

[点击购买](#)

或

[前往Kindle商店查看图书详情。](#)
